University of Dublin

TRINITY COLLEGE

Multiplayer Online Game using WebGL and WebSockets

Emma Carrigan
B.A.(Mod.) Computer Science
Final Year Project April 2015
Supervisor: Dr. Anton Gerdelan

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

DECLARATION

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university

_____ _____

Name Date

Abstract

The aim of this project was to investigate the new technologies, WebGL and WebSockets, and their suitability for creating an in-browser, multiplayer game.

These technologies were chosen because of their unique functionality in terms of online communication. WebGL allows for GPU powered graphics in the browser without the use of plug-ins, while WebSockets provide two-way real-time communication between server and client.

The result of this project shows that WebGL and WebSockets are capable of successfully implementing a 3D game with real-time player interaction. WebGL's hardware-accelerated graphics are on-par with the current industry standards and, as there is no need to install plug-ins, it makes any online application instantly more accessible to potential users. WebSockets also prove to be more than suitable for creating an in-browser game, as the WebScoket protocol not only matches competing technologies' abilities, but surpasses them, as it is the first protocol to provide two-way real-time communication across a single port over the web.

Acknowledgements

I'd like to thank Netsoc and Douglas Temple for providing a server for me to run this project, as well as answering my countless questions and fixing my many mistakes.

I'd also like to thank my supervisor, Anton Gerdelan, for the inspiration to do this project, his support and guidance throughout and for my resulting caffeine addiction.

Finally, I'd like to thank my friends and family, because you don't often get a chance to thank those who mean the most.

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Outline and Motivation

This project is implemented as a webpage and can be viewed at the following URL: `http://fiquem.netsoc.ie/graphics/FYP/graphics.php`

The purpose of this project is to investigate the new technologies, WebGL and WebSockets. WebGL is a new Javascript API that allows you to render graphics using the GPU and display the result in the browser, while the WebSocket protocol is a new protocol that defines a full-duplex communication channel which operates through a single port on a TCP connection. This report also discusses the use of the WebGL library, Three.js, as well as the server-side runtime environment, Node.js, with its library, Socket.IO, which facilitates WebSocket connections.

The investigation takes shape in the making of a multiplayer online game; a task which naturally requires real-time communication and high-quality graphics.

WebGL offers the unique ability to display hardware accelerated graphics in the browser. It's based on OpenGL, a current industry standard graphics library, and stays close to the OpenGL 2.0 ES specification, ensuring WebGL supplies functionality that is on-par with state of the art graphics technology[1].

The WebSocket protocol is unique in that it provides a communication channel between the client and the server that allows both to send and receive messages simultaneously - a full-duplex channel. This allows for real-time two-way communication between the browser and the server. WebSockets provide the added benefit of a handshake that resembles the HTTP protocol, enabling HTTP and WebSocket messages to be sent across the same port. This is a huge improvement on previous web protocols which,

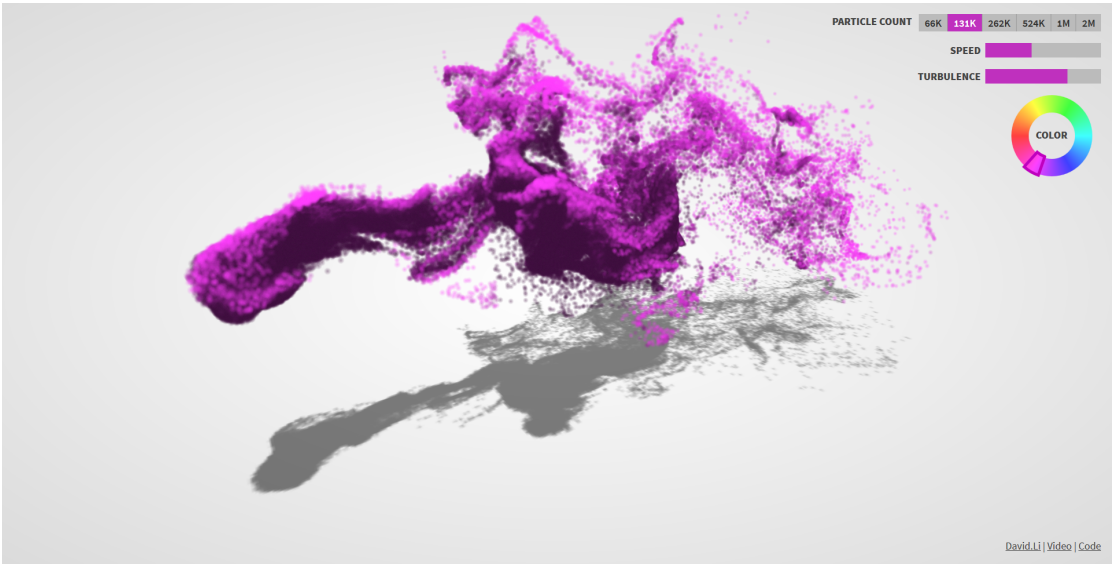

FIGURE 1.1: A Chrome Experiment in WebGL showing particle flow with user-controlled variable volume, speed and turbulence. Created by David Li[4].

at best, provided a half-duplex channel - both client and server could send and receive messages, but not simultaneously[2].

At the moment, applications written in languages similar to WebGL, such as OpenGL, face the issue of portability. Each application must be programmed differently depending on the platform on which it is to be released. WebGL provides a solution to this problem by using the browser as the platform. Using WebGL to make an application ensures that it can be run in a browser on any machine, provided that browser supports WebGL.

1.2 State of the Art

There have been a number of WebGL and WebSocket applications created in the last few years exploring the limits of these new technologies.

Perhaps the most famous examples of WebGL at the moment are the Chrome Experiments. Originally designed to test the limits of JavaScript and Google Chrome, the Chrome Experiments have become a showcase for new open-source, web-based technologies, including WebGL[3].

At the time of writing this report, there are 481 WebGL Chrome Experiments, with many of those experiments using Three.js, such as David Li's Volumetric Particle Flow experiment in Figure 1.1.

FIGURE 1.2: Reebok's Be More Human Experience - As well as the directed view that is shown here, a user can swap to a free rotation mode of the model in which the different parts of the brain are labelled.

Awwwards is a site for collecting and voting on websites based on their design and usability. Among others, it has a WebGL category. The sites listed here can either be purely artistic or somewhat commercial, giving an example of the diverse uses for WebGL[5].

One such example of a commercial use of WebGL is Reebok's Be More Human Experience, a section of which explains how exercise affects brain activity displayed on a brain rendered in WebGL[6]. (See Figure 1.2.) The use of 3D graphics in this instance not only provides a more interesting scene to look at, but also reinforces the ideas in the overlaid text, making the application clearer and more informative.

BrowserQuest is an MMO game experiment created by Little Workshop and Mozilla[7]. It uses a HTML5 `<canvas>` element to draw the game, which has a retro 2D graphics style, and WebSockets to support the real-time multiplayer gameplay, claiming the ability to support thousands of players simultaneously. It's server is implemented in Node.js[8].

The gameplay takes on a simple top-down RPG style and the aim of the game is to adventure and explore the world, completing quests in the process. (See Figure 1.3.)

Ironbane is an in-development MMORPG created using WebGL and WebSockets[9]. As of the writing of this report, it's in the pre-alpha stage with its most recent version is v0.4, released on the 7th of April 2015. It's backend is implemented in Angular.js and Meteor.js, recently updated from Node.js[10].

FIGURE 1.3: Players *Fiquem* and *fonzo_o* explore the outskirts of a village area of BrowserQuest.

FIGURE 1.4: Multiple players standing at the bottom of the Tower of Doom map of Ironbane.

Ironbane appears to be the only MMO using WebGL and WebSockets available. While there are other projects proposed or are currently being built, Ironbane is the only one that has a currently playable version and an active and interested community surrounding its development.

1.3 Literature

This section discusses the available literature surrounding the technologies used in this project.

2012, WebGL Beginner's Guide by Diego Cantor and Brandon Jones[11] gives a detailed overview of WebGL and graphics programming starting at an introductory level. It contains chapters dedicated to each aspect of creating a 3D scene with in-depth tutorials and explanations of the maths and graphics theory behind them.

The book assumes no knowledge and describes the JavaScript features that are encountered in the tutorials alongside the WebGL. It starts from the set-up involved in creating a WebGL project and proceeds to advanced topics, including post-processing and particle effects.

2012, WebGL Up and Running by Tony Parisi[12] is a more user-friendly introduction to WebGL, shirking some of the more hard-core theory and detail, instead opting to provide an overview of WebGL implemented with Three.js. This book also takes advantage of the fact that WebGL is rendered within a normal web page, giving examples of seamless integration of WebGL with HTML, CSS, JavaScript and jQuery.

The author of *WebGL Up and Running*, Tony Parisi, is renowned as a pioneer in the field of 3D standards for the web, having co-created the vector graphics markup languages VRML and X3D. Currently, he works on X3D and WebGL.

Chapter 2

Design & Implementation

2.1 Technologies

As has been made clear, this project is written using WebGL and WebSockets, however some further technologies were made use of during its implementation, some out of necessity and some purely for ease of use. This section will detail these technologies and their roles.

2.1.1 HTML, CSS, JavaScript and jQuery

Of course it is necessary to mention the basic technologies that are the foundation of this project. WebGL is a web technology and web technologies need a web page to function. This web page is implemented using very basic HTML that merely provides the necessary HTML tags that allow WebGL to draw, as well as an on-screen chat box.

However, more HTML is dynamically added to the page once the game is initialised. Player health, kills and deaths are displayed and styled using HTML and CSS and are updated using JavaScript.

JavaScript is also used to handle any in-game key-presses and mouse-clicks, while jQuery handles all key-presses that cause interactions with the web page outside of the game. jQuery is a JavaScript library that simplifies certain scripting tasks. In the case of this project, jQuery is used to select and manipulate HTML elements.

2.1.2 Three.js

Three.js is a JavaScript library for implementing 3D graphics using WebGL[13]. Although not an integral library to include in the production of a WebGL application, Three.js provides a number of inbuilt features that greatly reduce time and effort spent and make a program simpler and easier to read. It was because of this that the 3D graphics part of this project was written entirely in Three.js.

2.1.3 Ammo.js

Ammo.js is a JavaScript library for implementing realistic physics in a 3D environment. It works by defining a physics world with certain properties, e.g. gravity, and then defining objects in this world. These objects correlate with the objects drawn using Three.js. Once the physics scene is set up, Ammo.js calculates the next *step* in the physics world, i.e. the next position of each object, taking into account the forces that are acting upon it and the other objects it could possibly collide with. These coordinates are then used to draw the Three.js objects in the browser.

2.1.4 Node.js and Socket.IO

Node.js is a runtime environment that allows for server-side applications to be written in JavaScript. It's possible to implement a WebSocket server in a different programming language without Node.js, however it's convenient as the client-side of this project is written in JavaScript as well.

Socket.IO is a JavaScript library that provides bi-directional real-time communication between client and server primarily using the WebSocket protocol. It is essential to use Socket.IO in order to use the WebSocket protocol in a Node.js environment.

2.2 Game Client

The majority of the code in this project is written using Three.js and JavaScript to create the 3D world and implement gameplay. However, as described above, some features and functions of the client side of this project are implemented using HTML, CSS, jQuery and Ammo.js.

2.2.1 Basic Setup

In order to begin describing the game design process, it's necessary to set the scene. A WebGL project needs a web page and, as this project makes use of jQuery, Three.js and Ammo.js, it's necessary to include these libraries as well. There is also a Pointer Lock Control library that is a supplementary library for Three.js that will be used in section 2.2.3. For the most part, the code will be written within `<script>` tags after the `<body>`, as Three.js will append the graphics application that it creates to the `<body>` element and thus the `<body>` must exist before this occurs. (See Listing A.1.)

For Three.js to draw anything to the page, it first must define what needs to be drawn. It does this using a *Scene*. It's also necessary to define how to look at the objects that will be drawn in the *Scene*. This is done using a *Camera*. Finally, it needs an object to actually draw the *Scene*. This is called a *Renderer*. To draw an object, the object must be added to the *Scene* and then the *Renderer*'s *render()* function must be called, passing the *Scene* and *Camera* as parameters. This is usually done within an animation loop. (See Listing A.2.)

The described setup produces the most basic Three.js Scene with an animation loop, as can be seen in Figure 2.1.

It's useful to note that the *Camera* described in this process is actually a series of transformations on each vertex of each object in the scene in order to produce a scene viewed in a specific way. In this project, a perspective view is implemented with the camera positioned slightly above the origin (the origin is the point in the 3D world given the (x,y,z) coordinates (0,0,0)). To replicate this effect, every vertex in the scene is moved down to simulate a camera moving up, and vertices that are further away from the camera are scaled down, essentially being drawn closer together, to replicate perspective. An example of how perspective affects a scene can be seen in Figure 2.2.

2.2.2 Environment

To make this scene a little more interesting, an environment is required. The easiest way to add an impressive look with minimal effort is by adding a skybox.

A skybox is a cube with each of its sides given textures that blend seamlessly together such that, when the player looks around, it appears as if they are in the middle of the area depicted by the cube's textures.

This project uses textures created by Emil Persson, specifically the Måskonåive set of textures[14]. These textures conveniently come pre-labelled as which side of the

FIGURE 2.1: A Three.js Scene has successfully been set up, however nothing has been added to it yet. The screen instead displays Three.js's default background colour.

FIGURE 2.2: The cube on the left is viewed using an orthographic camera, that is its vertices are drawn at the same positions regardless of how far away they are from the camera. The cube on the right takes perspective into account and alters the position of vertices depending on how far away from the camera they are. Both of these cubes were below and to the left of the camera.

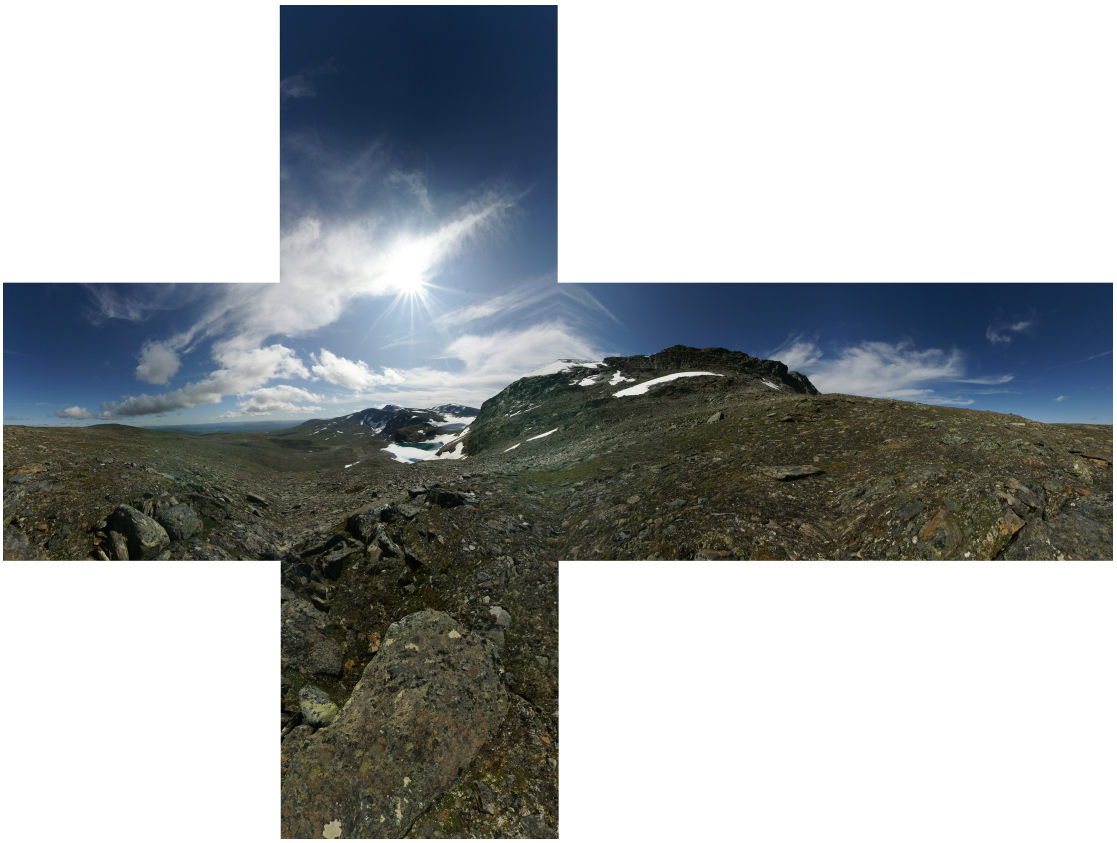

FIGURE 2.3: This image shows the six textures that will be mapped onto the skybox connected together such that, if they were folded, they would create the skybox implemented in this project.

cube they should be on, i.e. "posx", "negx", "posy", "negy", "posz", "negz". An "unwrapped" view of the cube texture can be seen in Figure 2.3. This, paired with the fact that Three.js supplies a function to texture each cube face separately given an array of textures, means creating an impressive looking environment is very simple. (See Listing A.3.)

Unfortunately, skyboxes are only aesthetic when viewed from afar, so the bottom of the skybox can't be used as the ground plane for the game without the visual effect being compromised. To solve this, there needs to be a ground plane created floating in the middle of the skybox to serve as an arena for the game.

The ground plane will be set at the origin while the camera is moved higher so that it can see the plane. For aesthetics' sake, the plane will be given a random, light colour. (See Listing A.4 and Figure 2.5.)

With a ground plane in place, the parallax effect comes into play. The parallax effect is the visual effect of closer objects moving quickly while further objects move slowly, giving the impression of distance and scale in an environment. The skybox in comparison to

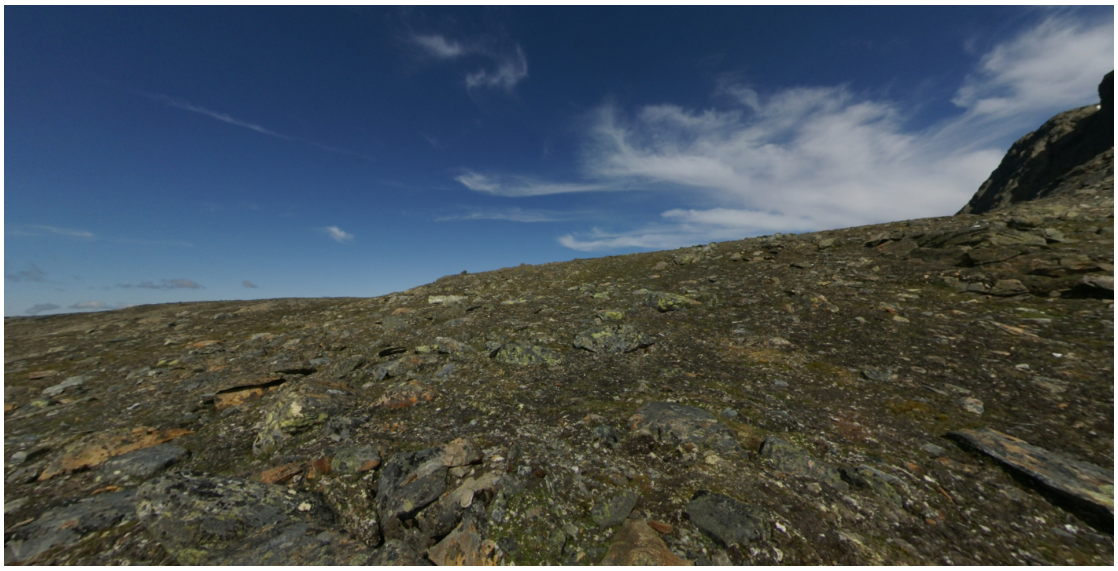

FIGURE 2.4: The scene including a skybox. A skybox is a large cube textured such that its sides blend seamlessly together, creating a realistic environment.

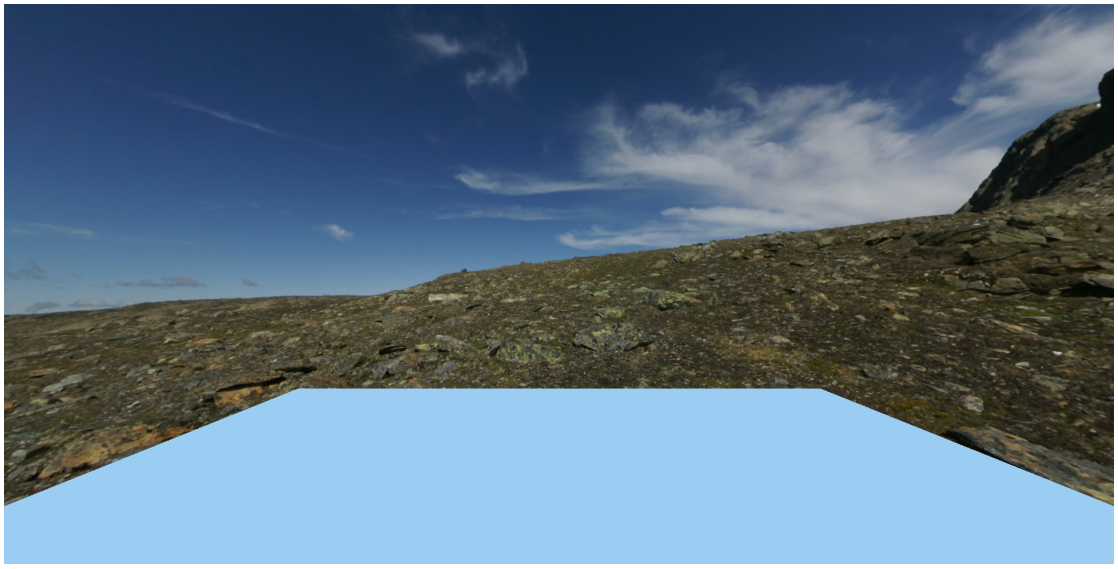

FIGURE 2.5: The scene including a ground plane.

the ground plane creates this parallax effect which can be explored after implementing camera movement as described in section 2.2.3.

A skybox is an implementation of cube mapping, which is the technical term for texturing each of a cube's side with images such that they produce a seamless environment. Cube mapping, in turn, can be used for more complex visual techniques, such as creating a reflection effect on an object placed within the cube-mapped environment[15].

FIGURE 2.6: The scene viewed from a different angle after moving the camera.

2.2.3 Player Controls

Three.js provides a control object that be defined to operate the camera under a given control system. For a first-person game, it's expected that moving the mouse will move the camera to look in that direction without changing position. In Three.js, this is called *PointerLockControls()*.

As the name suggests, to use this control system it's necessary to first obtain pointer lock. This is done by making a pointer lock request when the player clicks within the browser window. Currently, pointer lock isn't standardised so the request must be tailored for each browser. (See Listing A.5.)

As well as moving the camera, a first-person control system should include the ability to move forward, backward, left and right and to jump. This is done by adding event handlers for button presses in JavaScript which complete actions depending on which button is pressed. Specifically, in the case of this game, the *W*, *S*, *A* and *D* keys move the player forward, backward, left and right respectively and *Space* causes the player to jump. (See Listings A.6, A.7.)

2.2.4 Box Physics

In this project, Ammo.js was implemented but not fully integrated into the gameplay. When a player joins the game, a part of the initial setup of the game spawns a number of boxes at random offsets in mid-air in the middle of the map. These boxes use Ammo.js

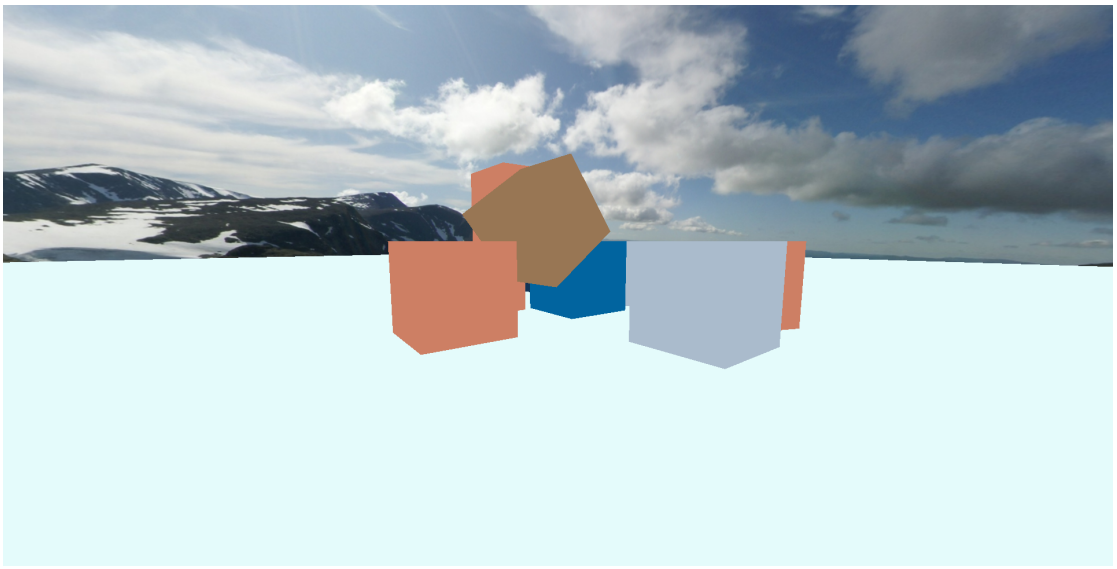

FIGURE 2.7: The physics boxes, settled after having fallen from their initialised positions.

to calculate how exactly they fall, taking into account their collisions with each other and the ground plane that they cannot fall past. (See Listing A.8 and Figure 2.7 as well as the file 'js/worker.js' included on the accompanying CD due to its large size - all code adapted from Ammo.js example code[16].)

As the player isn't included in the physics world that Ammo.js uses to position the boxes, the player cannot collide with them. It's also important to note that the boxes are initialised by a player when they connect and each player's boxes' positions aren't shared with other players, so every player sees a slightly different set of boxes.

Ammo.js provides a degree of physical realism that's difficult to replicate and adds a great amount of value to the game. Further work on this project would include a fuller implementation of Ammo.js.

2.2.5 Player Stats

The basic idea of a first-person shooter game is an arena with multiple players shooting each other to get kills. In order for this to happen, each player needs an amount of health that will decrease when they get shot. On top of this, it's also useful for the player to keep track of the number of times they have killed another player and the number of times they have died.

These are stored simply as JavaScript variables. However, it is polite to allow the player to view information about themselves at any time. This is achieved by displaying the

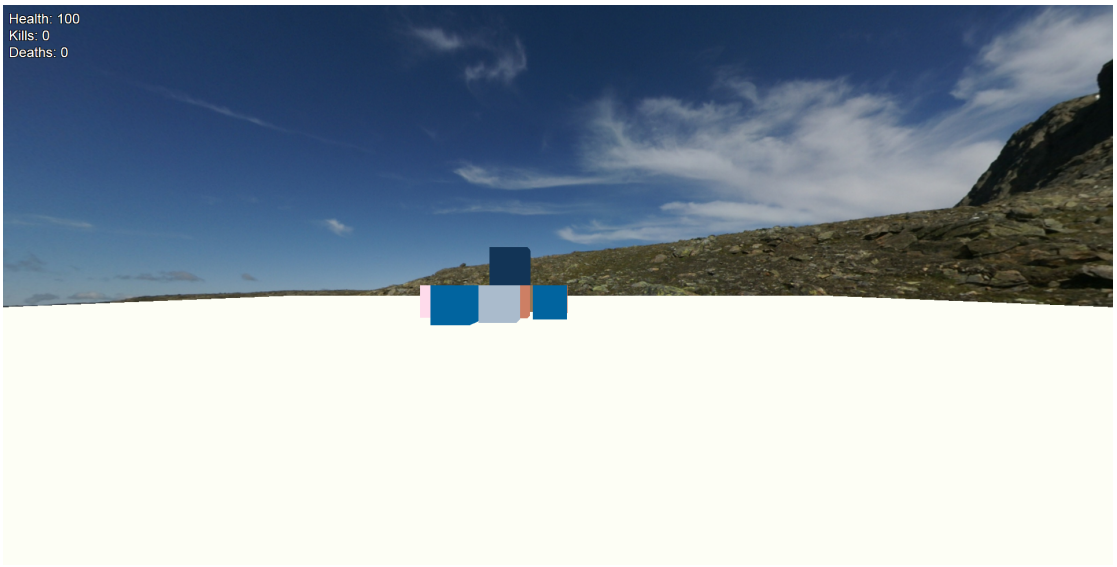

FIGURE 2.8: The player's Health, Kills and Deaths are displayed on-screen using HTML, CSS and JavaScript. This display isn't connected to WebGL or Three.js.

player information on-screen as 2D text over the 3D graphics. (Listing A.9, Figure 2.8.) This information can then be accessed and updated using an update function that is called any time a player's information changes. (Listing A.10.)

2.2.6 Chat

This section marks the beginnings of this project's multiplayer capabilities. A chat feature isn't necessary for this game, but it provided a simple testing ground for WebSockets and added some entertainment value to the game, namely a forum for users to announce their displeasure at the incomplete status of the game.

This project's chat feature is heavily based off Martin Sikora's *Node.js & WebSocket - Simple chat tutorial*[17]. The complete code to implement a chat client will not be included here, but can be found at 'js/multiplayer.js' on the accompanying CD. Simplified code to set up a WebSocket connection can be seen in Listing A.11. Furthermore, large changes were made to the appearance of the chat box in the game, as well as minor changes to how the chat messages are relayed from client to server. These can be seen in Listings A.12 and A.13, with the rendered result shown in Figure 2.9.

The most notable change is the ability to show or hide the chat box. This is necessary as the chat takes up a large amount of screen space in order to display an appropriate number of messages before the user must scroll up to see more, and as the chat is a secondary feature, permanently displaying it would affect a user's interaction with the game.

FIGURE 2.9: The chat box is slightly transparent in order to be less intrusive while still being large enough to be comfortable to use.

The chat box is written in HTML and CSS and the ability to toggle the chat open or closed using the *Enter* key is written in jQuery. (See Listing A.14.)

This chat implementation is continued in section 2.3, where the server-side implementation is described.

2.2.7 Sending and Receiving Player Information

Now that the connection to the server has been established (See Figure 2.10), it's time to send the other relevant player information to the server which will the broadcast it to all connected players. The server-side implementation of this is described in section 2.3.

At the moment, the relevant player information to be sent is the player's position, which will allow other players to see the player, and the player's kills and deaths, which we will display on a scoreboard. Sending these messages is trivial and closely resembles how the chat messages in the previous section were sent. (See Listing A.15.)

It's more interesting to look at what happens when a client receives messages from the server. In the case of player coordinates, a number of things happen. First, the client must determine whether the player coordinates it is receiving are for a player that has been seen before. If not, the client must create a new player mesh (in the case of this project, a pink box). If the player has been seen before, it will already have a mesh created. After that, whether or not the coordinates are for a new player or not,

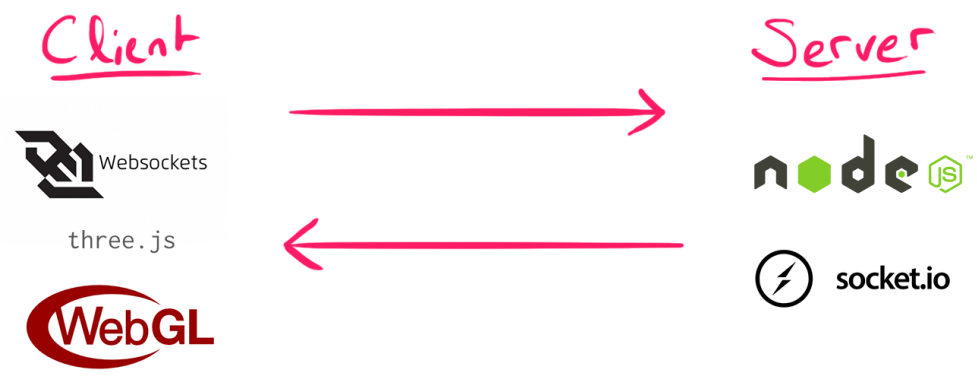

FIGURE 2.10: A simple visualisation of the two-way communication between the client and the server and the technologies used on both sides.

the player mesh is drawn at the coordinates received from the server. These player coordinates and player meshes are stored in associative arrays using the player name as the array index. (See Listing A.16.)

Currently, nothing is in place to deal with player kill and death information when it is received. Using some simple HTML, CSS and jQuery, a basic scoreboard can be created to display this information. For the purposes of this game, the scoreboard will initially be hidden, like the chat. The following code snippets show the receipt and storage of player information to be displayed, the construction of the scoreboard and the ability to toggle it open or closed. See Listings A.17, A.18 and A.19 respectively, and Figure 2.11 for a rendered result.

Now that the client can deal with incoming player information, the game is starting to seem a bit more multiplayer!

2.2.8 Player Nametags

It's often desirable to know what players you're looking at. This is most easily achieved by displaying a player's name above their player model. In this project, this is done by floating an invisible plane above each player and texturing that plane with the player's name. As player names vary, so must the texture. For a variable texture, it's necessary to create the texture on the fly, which can be done using the HTML `<canvas>` element. By creating a canvas and declaring it to be a 2D canvas rather than a 3D WebGL canvas, it can be written onto and cleared as needed, and the text on the canvas can be accessed at any time and applied to a plane as a texture. As well as that, the `<canvas>` element can remain hidden while the information about what's written on it can still be accessed.

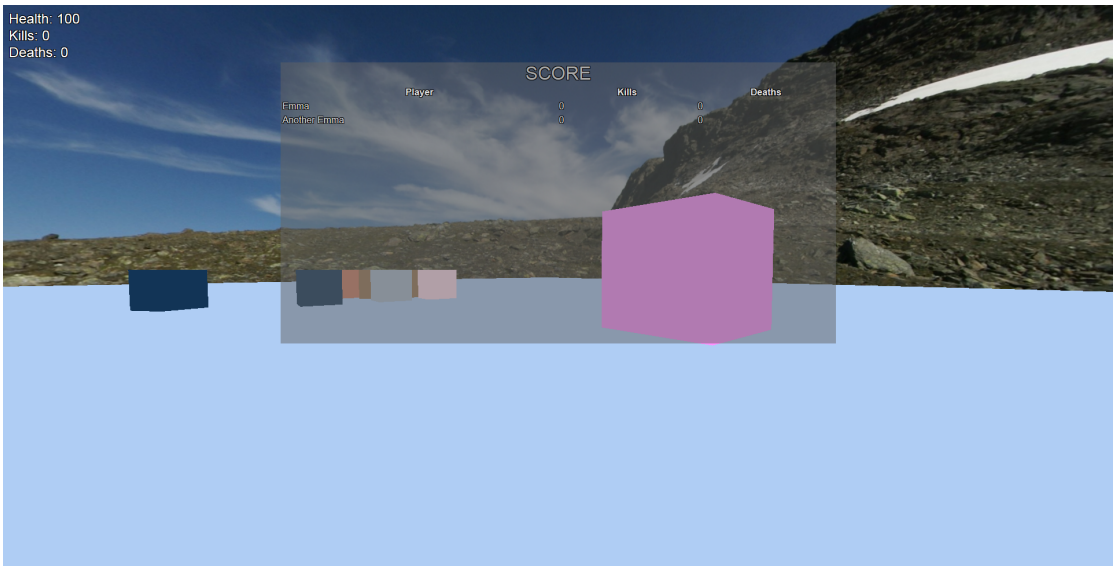

FIGURE 2.11: The scoreboard displays all current connected players' names, kills and deaths. This screenshot is the first-person view from one of those players. The other can be seen on the right behind the scoreboard. The players are depicted as large, pink boxes.

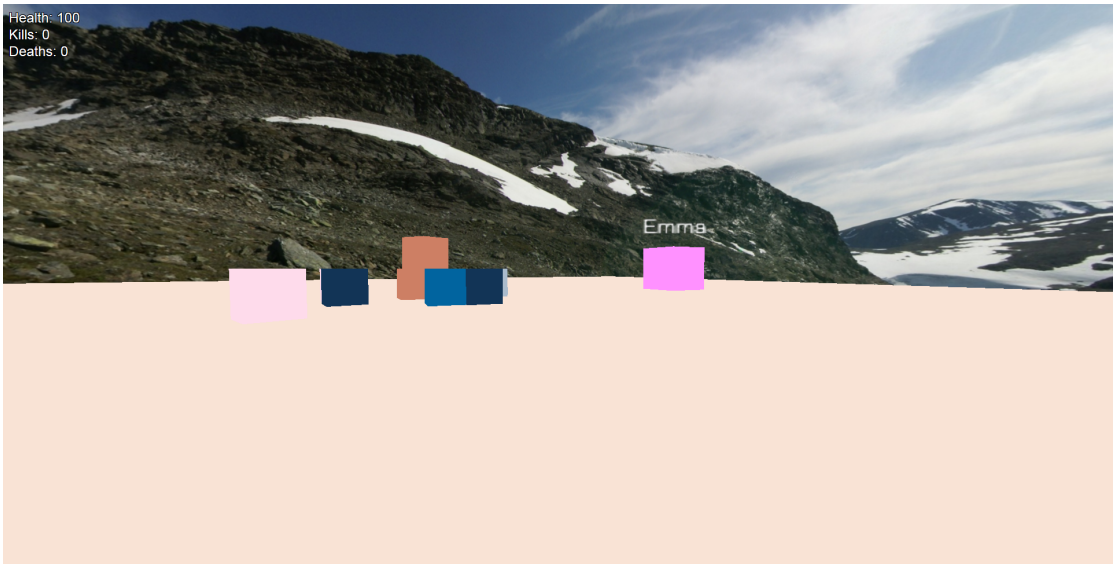

FIGURE 2.12: A player's name floats above their model and rotates such that the nametag plane is always perpendicular to the player looking at it.

The initialisation of the canvas can be seen in Listing A.20, while the initialisation and drawing of the nametag planes can be seen in Listings A.21 and A.22. The final result can be seen in Figure 2.12.

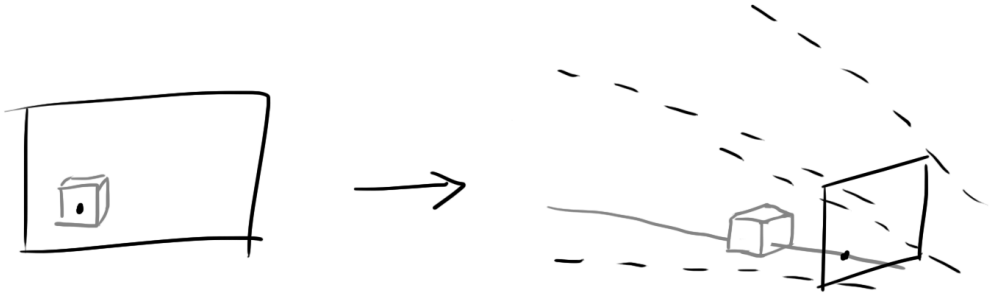

FIGURE 2.13: On the left is a view of a 3D cube after it has been "projected" to the 2D screen. The black dot represents the point on this screen through which a ray will be cast. On the right is a visualisation of that point on the 2D screen being "unprojected" so that it appears as a line in the 3D world. The line intersects with the cube, as is expected from the 2D image on the left where the point is directly on top of the cube.

2.2.9 Shooting, Killing, Dying

The most important features of a first-person shooter game is the ability to shoot other players and to record the number of kills a player gets, i.e. the number of times their shot has caused the death of another player.

The shooting mechanism is created using raycasting, or more specifically, picking. Picking is the process of holding a mouse pointer over the 2D image on a screen and identifying the 3D object that the mouse is covering. In order to this, a virtual "ray" is cast through the mouse into the 3D scene, starting at the position of the camera. At the moment the mouse is placed on the screen, the "ray" is a point on the screen, a point on a 2D image. However, by reversing the steps taken to draw an object to the screen (3D -> 2D) it's possible to find the line in the 3D world that this point in the 2D world represents (2D -> 3D)[18]. Figure 2.13 gives a basic visualisation of the ray creation process.

Once the ray is in place, the game checks for intersections between the ray and every other player. If any intersections are found, it takes the first one, i.e. the first player it hit, and discards the rest. By discarding the later intersections, it is ensured that only one player is hit and, if any other players are behind this player and the ray intersects with them, they will not be affected. This information is then stored as a *shot* and sent to the server, which then alerts the player that was shot that they need to decrease their health. (See Listing A.23.)

When a player's health reaches zero, they die. They send a message informing the server of the last player to shoot them. The server then tells that player to increment their kills. An implementation of the messages involved in shooting, killing and dying can be seen in Listing A.24.

2.3 Game Server

The server for this project is far simpler than the client. It is a single file written in JavaScript and run using Node.js with the Socket.IO library. Again, the basis for this code is Martin Sikora's *Node.js & WebSocket - Simple chat tutorial*[17]. The code will not be quoted here but can be found in 'chat-server.js'.

The server, for the most part, takes on the same form as the client, the main difference being the setup. As a WebSocket connection is established on a HTTP connection, the server must listen for HTTP messages requesting a WebSocket connection. For this purpose alone, a HTTP server is defined. The WebSocket server is then defined with the HTTP server as an attribute and, after defining an event handler for a 'request' message, any WebSocket connection requests can succeed and the WebSocket server begins to look very similar to the client.

From this point on, the server can receive messages. Specifically, this server is programmed to identify JSON messages. JSON stands for JavaScript Object Notation and a JSON message takes the form of a list of attribute-value pairs, e.g. a message could simply be "type: chat". The messages this server expects will always include a "type" attribute.

On receipt of a valid JSON message, i.e. a JSON message with a "type" attribute, the server detects its type and reacts accordingly, e.g. storing a player's coordinates in the coordinates array on receipt of a message of type *player_coords*. The server also sends messages at given intervals, just like the client. Specifically, the server broadcasts all players' coordinates and current score, as well as every shot and death that has happened in the last interval.

Importantly, the server also broadcasts a notification of a player disconnecting. This is so clients can stop tracking information about that player.

Chapter 3

Discussion

3.1 Three.js

Possibly the most drastic design choice that was made was to use Three.js. Although the most obvious reasons to use it are its rich library of inbuilt features and intuitive API compared to pure WebGL, these were not deciding factors for its use in this project. The decision to use Three.js came about after a large portion of the game had already been implemented, when the physics library Ammo.js became involved.

Ammo.js was enticing as it provided demos of realistic collisions and seemed not too difficult to implement, however there are very few demos of Ammo.js available and the most concise and clear of those used Three.js. Without a clear idea of how Ammo.js worked at the time, the decision was made to rebuild the project using Three.js so that Ammo.js could be integrated.

After switching to Three.js it became difficult to stop using it. Certain small features sped up even the smallest of tasks enormously. One such feature was the availability of inbuilt primitive meshes, i.e. cubes, spheres, etc., which, in WebGL, the programmer would have to create in 3D modelling software themselves, save in a certain format and import using self-written code or a code provided by a third party. On top of this, the code to draw a single mesh in WebGL is a couple of lines or more, depending on whether or not there are textures involved or transformations (translation, rotation, scaling) performed on that mesh.

Similarly, Three.js supplies textures for those meshes which, in WebGL, would again have to be created and imported by the programmer separately before they would be available to apply to a mesh.

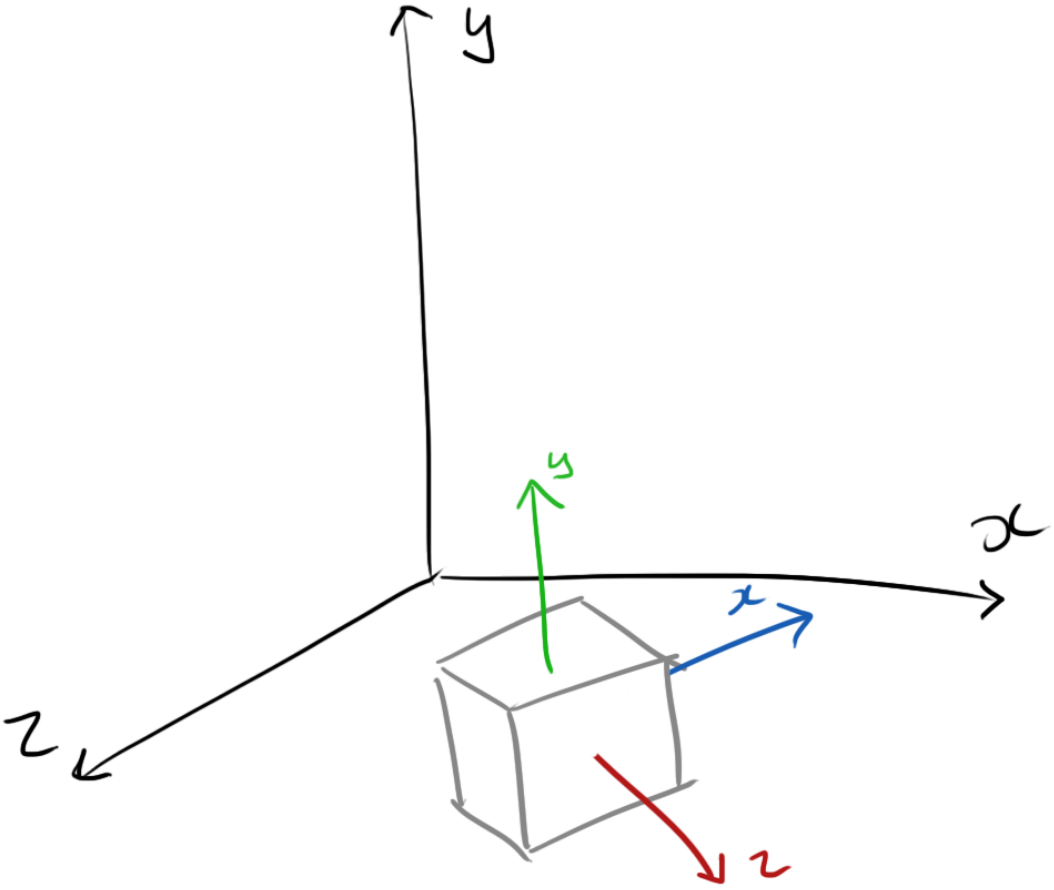

FIGURE 3.1: The world xyz-axes are drawn in black. Thiese are unchanging axes which are the basis for all objects' placement in the 3D world. Within the world, a grey cube has been rotated slightly such that its local axes no longer line up with the world axes. The cube's local axes are drawn in colour, specifically x in blue, y in green and z in red.

Three.js contains a Controls object that can define how a certain object is controlled - in the case of this project, pointer lock controls were applied to the camera. This meant that, once pointer lock was obtained by the user clicking in the browser window and accepting the pointer lock, the camera would rotate to look in the direction of the mouse movement, just like is expected of a first-person styled game. On top of this, the Controls object stores the local xyz-axes of the camera, making the usually finicky task of moving along those axes quite simple. (See Figure 3.1.)

Three.js has dominated the WebGL scene by being the oldest and most popular library available. There is at least one alternative, however, which appears promising enough to deserve to be considered an alternative. Babylon.js is a newer library, created in 2013, and which has the advantage of being originally designed as a Silverlight game engine, promising a heavily game development influenced API[19]. It has a number of

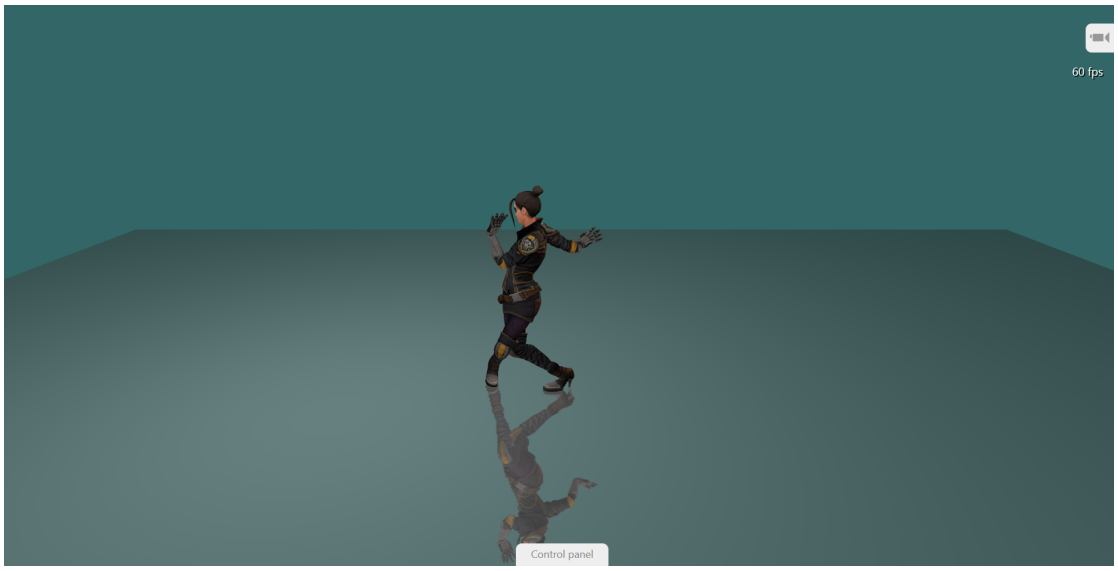

FIGURE 3.2: One of the many Babylon.js demos available on the official Babylon.js site[20]. This demo depicts a model dancing with impressively realistic movements.

demos on its site, not quite as extensive as the Chrome Experiments, but impressive nonetheless[20]. (See Figure 3.2.)

3.2 Why use WebGL?

In terms of 3D web technology, WebGL is the forerunner. Alternatives include Unity 3D, Flash and Silverlight. WebGL's most obvious advantage over these is that it doesn't require a plug-in, which Flash and Silverlight require, or the installation of a web-player, which Unity requires. Currently, Unity is in the process of adding a web export function to their engine which will compile any Unity application to WebGL[21].

WebGL also has the advantage of being available on mobile devices, which Flash and Silverlight aren't. Unity can be compiled to run on mobile devices, however in this case it can't be directly accessible through a web page[22].

In terms of cross-platform technology with minimal barriers to use, WebGL comes out on top.

3.3 Why use WebSockets?

As was mentioned in section 1.1, the WebSocket protocol is the first protocol to allow a full-duplex communication channel across a single socket over the web. In a comparison

to previously implemented half-duplex channels, WebSockets can provide a huge reduction in HTTP header overhead as well as a reduction in latency - one article specifying a 500:1 reduction in overhead and 3:1 reduction in latency[2]. On top of this, WebSockets provide a reduction in complexity, as the protocol has an easy to use API.

3.4 Technology Summary

Below is a table summarising the values of the technologies discussed as they were encountered in this project. The table ranks the technologies in terms of Difficulty to use, previous Experience required, development Time required, Dependencies or other requirements, Reliability, Performance and availability and usefulness of Documentation. It ranks these fields either low, medium or high.

Technology	**Difficulty**	**Exp.**	**Time**	**Depend.**	**Reliability**	**Perf.**	**Doc.**
WebGL	high	high	high	low	high	high	high
WebSockets	low	low	low	low	high	high	high
Three.js	low	low	medium	low	high	high	high
Ammo.js	high	high	high	medium	high	high	low

WebGL and WebSockets are powerful technologies that are highly recommended for any online graphical applications that require client-server communication. However, while WebSockets are simple to implement no matter the language preference of the developer, WebGL is complex and difficult to use without previous experience in computer graphics. A programmer should have a knowledge of the hardware pipeline and the different shaders involved in that in order to create a WebGL application.

Three.js provides a solution for this problem. With a simple and intuitive API, Three.js can be used to implement graphics using WebGL without any previous experience.

Although the Ammo.js API is simple, the theory behind it is not and it's necessary to spend a lot of time on understanding simple examples in order to even partially integrate Ammo.js into a project. On top of this, Ammo.js isn't very well documented. It's updated frequently and doesn't cater to a wide audience, so tutorials and demos are rare and often outdated.

FIGURE 3.3: A still from the video featured in Mozilla's article about Unity 5's WebGL 2 export function[24].

3.5 Future of WebGL

The Khronos Group that created WebGL are currently developing WebGL 2, which will be based on OpenGL ES 3.0[23]. This advancement is highly anticipated by Mozilla and Unity, with Firefox already able to support the experimental WebGL 2 and Unity 5 engine including experimental support for export of projects as WebGL 2[24]. (See Figure 3.3.)

Appendix A

Code Segments

```
<html>
        <head>
                <script type="text/javascript" src="js/ammo.js"></script>
                <script type="text/javascript" src="js/three.min.js"></script>
                <script type="text/javascript"
    src="js/controls/PointerLockControls.js"></script>
                <script src="//code.jquery.com/jquery-2.1.3.min.js"></script>
        </head>
        <body style="margin: 0; overflow: hidden;">
        </body>
        <script>

                ⋮


        </script>
<html>
```

Listing A.1: HTML page importing libraries. The `<body>` is styled using CSS so that it fills the browser window and no more, i.e. there will be no margin between the `<body>` and the window border and, if any element appears outside this frame, no scrollbar will appear to allow us to reach it.

```
var scene, camera, renderer;

init();
animate();

function init() {
        scene = new THREE.Scene();
        camera = new THREE.PerspectiveCamera( 75, window.innerWidth /
    window.innerHeight, 1, 20000 );

                ⋮

        container = document.createElement( 'div' );
        document.body.appendChild( container );

        renderer = new THREE.WebGLRenderer();
        renderer.setSize( window.innerWidth, window.innerHeight );
        container.appendChild( renderer.domElement );
}

function animate() {
        requestAnimationFrame( animate );

                ⋮

        renderer.render( scene, camera );
}
```

LISTING A.2: Setting up a Three.js *Scene* and animation loop. The container for the *Renderer* in *init()* is appended to the `<body>` element, which is why this code snippet appears after `<body>` has been declared. The *Renderer* is defined as a *WebGLRenderer()* as is possible for Three.js to render non-WebGL. In *animate()*, the *requestAnimationFrame()* function is used with *animate()* as the parameter to create the animation loop.

```
function init(){
        ⋮
        // skybox
        var imagePrefix = "textures/Maskonaive/";
        var directions  = ["posx", "negx", "posy", "negy", "posz", "negz"];
        var imageSuffix = ".jpg";
        var skyGeometry = new THREE.CubeGeometry( 10000, 10000, 10000 );
        var materialArray = [];
        for (var i = 0; i < 6; i++)
                materialArray.push( new THREE.MeshBasicMaterial({
                        map: THREE.ImageUtils.loadTexture( imagePrefix +
    directions[i] + imageSuffix ),
                        side: THREE.BackSide
                }));
        var skyMaterial = new THREE.MeshFaceMaterial( materialArray );
        var skyBox = new THREE.Mesh( skyGeometry, skyMaterial );
        scene.add( skyBox );
        ⋮
}
```

Listing A.3: Code to implement a skybox in Three.js given the texture location, names and suffix.

```
function init(){
        ⋮
        // ground
        geometry = new THREE.PlaneGeometry( 500, 500 );
        geometry.applyMatrix( new THREE.Matrix4().makeRotationX( - Math.PI / 2 )
    );
        var groundColour = new THREE.Color().setHSL( Math.random(), 0.75,
    Math.random() * 0.25 + 0.75 );
        for ( var i = 0, l = geometry.faces.length; i < l; i++ ) {
                geometry.faces[i].color = groundColour;
        }
        material = new THREE.MeshBasicMaterial( { vertexColors: THREE.FaceColors
    } );
        mesh = new THREE.Mesh( geometry, material );
        scene.add( mesh );
        ⋮
}
```

Listing A.4: Code to implement a ground plane in Three.js. A plane is created, then rotated to be lying flat on the xz-axes rather than the zy-axes. A colour is randomly generated then assigned to each of the plane's faces. Finally, a material is created using these face colours, a mesh is created using the plane shape and the material, and this mesh is added to the scene.

```
// Ask the browser to lock the pointer
document.body.requestPointerLock = document.body.requestPointerLock ||
    document.body.mozRequestPointerLock ||
    document.body.webkitRequestPointerLock;
document.body.onclick = function() {
        document.body.requestPointerLock();
}

var pointerlockchange = function ( event ) {
        if ( document.pointerLockElement === document.body||
    document.mozPointerLockElement === document.body||
    document.webkitPointerLockElement === document.body) {
                controls.enabled = true;
        } else {
                controls.enabled = false;
        }
}

// Hook pointer lock state change events
document.addEventListener( 'pointerlockchange', pointerlockchange, false );
document.addEventListener( 'mozpointerlockchange', pointerlockchange, false );
document.addEventListener( 'webkitpointerlockchange', pointerlockchange, false );

function init(){
        .
        .
        .
        controls = new THREE.PointerLockControls( camera );
        controls.getObject().position.y = 100;
        controls.getObject().position.z = 100;
        scene.add( controls.getObject() );
        .
        .
        .
}
```

LISTING A.5: Code to request Pointer Lock in JavaScript and to attach *PointerLockControls()* to the current camera and add these controls to the scene. The controls then move the camera higher to get a better view of the scene.

```
var moveForward = false;
var moveBackward = false;
var moveLeft = false;
var moveRight = false;

function init(){
        ⋮
        var onKeyDown = function ( event ) {
                switch ( event.keyCode ) {
                        case 87: // w
                                moveForward = true;
                                break;
                        case 65: // a
                                moveLeft = true; break;
                        case 83: // s
                                moveBackward = true;
                                break;
                        case 68: // d
                                moveRight = true;
                                break;
                        case 32: // space
                                if ( canJump === true ) velocity.y = 300;
                                canJump = false;
                                break;
                }
        };

        var onKeyUp = function ( event ) {
                switch( event.keyCode ) {
                        case 87: // w
                                moveForward = false;
                                break;
                        case 65: // a
                                moveLeft = false;
                                break;
                        case 83: // s
                                moveBackward = false;
                                break;
                        case 68: // d
                                moveRight = false;
                                break;
                }
        };

        document.addEventListener( 'keydown', onKeyDown, false );
        document.addEventListener( 'keyup', onKeyUp, false );
        ⋮
}
```

Listing A.6: Code to signal whether movement in a certain direction should be allowed determined by key-press.

```
var velocity = new THREE.Vector3();
var prevTime = performance.now();
var moveForward = false;
var moveBackward = false;
var moveLeft = false;
var moveRight = false;

function animate() {
        requestAnimationFrame( animate );

        var time = performance.now();
        var delta = ( time - prevTime ) / 1000;

        // simulated forces
        velocity.x -= velocity.x * 10.0 * delta;
        velocity.z -= velocity.z * 10.0 * delta;
        velocity.y -= 9.8 * 100.0 * delta; // 100.0 = mass

        if ( moveForward ) velocity.z -= 1500.0 * delta;
        if ( moveBackward ) velocity.z += 1500.0 * delta;
        if ( moveLeft ) velocity.x -= 1500.0 * delta;
        if ( moveRight ) velocity.x += 1500.0 * delta;

        controls.getObject().translateX( velocity.x * delta );
        controls.getObject().translateY( velocity.y * delta );
        controls.getObject().translateZ( velocity.z * delta );

        if ( controls.getObject().position.y < 10 ) {
                velocity.y = 0;
                controls.getObject().position.y = 10;
                canJump = true;
        }

        prevTime = time;

        renderer.render( scene, camera );
}
```

LISTING A.7: Code showing how movement is dealt with. Notice Three.js allows movement along the camera's local axes very simply through its *controls.getObject()* functions. Notice also the forces on each axis, simulating gravity on the y-axis and friction on the x- and z-axes. There is also an absolute minimum position set such that the player will never fall past a certain point.

```
var physicsWorker = new Worker('js/worker.js');
var colours = [0x123456, 0x987654, 0xaabbcc, 0xfedbeb, 0x01649f, 0xcd7f64];
var boxSize = 10;
var boxes = [];
var NUM = 10;
⋮

function init(){
        ⋮
        // boxes
        for (var i = 0; i < NUM; i++) {
                material = new THREE.MeshBasicMaterial({ color:
    colours[Math.floor((Math.random()*100)%6)] });
                geometry = new THREE.BoxGeometry( boxSize, boxSize, boxSize);
                boxes[i] = new THREE.Mesh( geometry, material );
                scene.add(boxes[i]);
        }

        // Worker
        physicsWorker.onmessage = function(event) {
                var data = event.data;
                if (data.objects.length != NUM) return;
                for (var i = 0; i < NUM; i++) {
                        var physicsObject = data.objects[i];
                        boxes[i].position.x = physicsObject[0];
                        boxes[i].position.y = physicsObject[1];
                        boxes[i].position.z = physicsObject[2];
                        boxes[i].quaternion.x = physicsObject[3];
                        boxes[i].quaternion.y = physicsObject[4];
                        boxes[i].quaternion.z = physicsObject[5];
                        boxes[i].quaternion.w = physicsObject[6];
                }
                currFPS = data.currFPS;
                allFPS = data.allFPS;
        };
        physicsWorker.postMessage(NUM);
        ⋮
}
```

LISTING A.8: Code showing how to integrate the physics library Ammo.js into a Three.js project. In this example, all the Ammo.js code is in the 'js/worker.js' file. The code here receives the next position for each box from the physics worker and positions the boxes appropriately.

```
var myHealth = 100;
var myKills = 0;
var myDeaths = 0;

function init(){
        ⋮
        init_player_stats();
        ⋮
}

function init_player_stats () {
        var stats = document.createElement('div');
        stats.id = 'player_stats';
        stats.style.position = 'absolute';
        stats.innerHTML = "Health: " + myHealth + "<br>Kills: " + myKills +
    "<br>Deaths: " + myDeaths;
        stats.style.top = 10 + 'px';
        stats.style.left = 10 + 'px';
        stats.style.fontFamily = 'helvetica';
        stats.style.fontSize = 24 + 'px';
        stats.style.textShadow = "\
                -1px -1px 0 #000,\
                1px -1px 0 #000,\
                -1px 1px 0 #000,\
                1px 1px 0 #000";
        stats.style.color = "#FFFFFF";
        document.body.appendChild(stats);
}
```

Listing A.9: This code initialised the player's health, kills and deaths and displays them on-screen, using JavaScript to create a HTML `<div>` element and style the text inside using CSS.

```
function display_player_stats () {
        if (!myHealth) respawn ();
        else {
                var stats = document.getElementById('player_stats');
                stats.innerHTML = "Health: " + myHealth + "<br>Kills: " +
    myKills + "<br>Deaths: " + myDeaths;
                }
        }
}

function respawn () {
        controls.getObject().position.x = 0;
        controls.getObject().position.y = 100;
        controls.getObject().position.z = 100;
        velocity.x = 0;
        velocity.y = 0;
        velocity.z = 0;
        myHealth = 100;
        myDeaths++;
        display_player_stats();
}
```

LISTING A.10: This code updates the player's health, kills and deaths in the on-screen display. It also includes a *respawn()* function that is called on the player's health reaching zero. The function resets all player info apart from kills, which it ignores, and deaths, which it increments.

```
window.WebSocket = window.WebSocket || window.MozWebSocket;

if (!window.WebSocket) {
        content.html($('<p>', { text: 'Sorry, but your browser doesn\'t ' +
    'support WebSockets.'} ));
        input.hide();
        $('span').hide();
        return;
}

var connection = new WebSocket('ws://servername:8080');

connection.onopen = function () {
        ⋮
};

connection.onerror = function (error) {
        ⋮
};

connection.onmessage = function (message) {
        ⋮
};
```

Listing A.11: As Firefox supplies its own built-in WebSocket, it's necessary to first check if the user is using Firefox and change what type of WebSocket to work with appropriately. As well as that, it's possible the browser may not support WebSockets at all. If it does, however, a WebSocket connection can be established very simply and the necessary event handlers can be defined, namely *onopen*, *onerror* and *onmessage*. It is also possible to define an *onclose* event handler.

```
<head>
        ⋮
        <style>
                #content
                {
                        font-family: helvetica;
                        font-size: 15px;
                        color: #FFFFFF;
                        text-align: left;
                        position: absolute;
                        bottom: 65px;
                        height: 30%;
                        width: 75%;
                        overflow-y: scroll;
                        overflow-x: hidden;
                        background: rgba(0, 0, 0, 0.3);
                        left: 50%;
                        transform: translate(-50%, 0);
                }
                #input
                {
                        position: absolute;
                        bottom:20px;
                        padding:15px;
                        width:75%;
                        left: 50%;
                        transform: translate(-50%, 0);
                }
        </style>
</head>

<body style="margin: 0; overflow: hidden;">
        <div id="Websockets" style="display: none;">
                <div id="content"></div>
                <input type="text" id="input" placeholder="Enter name, press
    enter, begin chatting!">
        </div>
</body>
```

Listing A.12: The chat box is displayed inside a `<div>` element that is initially toggled hidden. Inside this, we have a *content* `<div>` that displays the chat messages and an `<input>` element that allows the player to send their own messages to the server.

```
connection.send(JSON.stringify({type: 'chat', message: 'Hi!'}));
```

Listing A.13: This line sends to the server a JSON message informing the server that the *type* of message is 'chat' and the *message* is 'Hi!'. The original code didn't include a *type* attribute, as the original purpose of this program was purely as a chat client. This project plans to expand the types of messages sent to include player information in order to properly implement a multiplayer online game.

```
var enterDown = false;
$(document).keydown(function (event) {
        if (event.which == 13) {
                if(!enterDown) {
                        enterDown = true;
                        $(document.getElementById("Websockets")).toggle();
                        var elem = document.getElementById('content');
                        elem.scrollTop = elem.scrollHeight;
                        $("#input").focus();
                }
                event.stopImmediatePropagation();
                event.stopPropagation();
                event.preventDefault();
        }
});

$(document).keyup(function (event) {
        if (event.which == 13) enterDown = false;
});
```

LISTING A.14: This code identifies an *Enter* key press in the browser window and opens the chat box if it is closed or vice versa.

```
setInterval(function() {
        connection.send(JSON.stringify({type: 'player_coords', coords:
    cam_pos}));
        connection.send(JSON.stringify({type: 'player_kd', kd: [myKills,
    myDeaths]}));
}, 10);
```

LISTING A.15: This code sends two messages to the server at an interval of 10ms. The interval time was chosen to allow for near-real-time communication without flooding the server with messages.

```
connection.onmessage = function (message) {
        ⋮
        if (json.type === 'player_coords') {

                // If first instance, create player mesh
                var new_player = true;
                for (var x in player_meshes){
                        if (player_meshes.hasOwnProperty(x) && x ===
    json.data.name){
                                new_player = false;
                        }
                }
                if (json.data.name != "false" && json.data.name != myName &&
    new_player) {
                        console.log(json.data.name);
                        var g = new THREE.BoxGeometry( 20, 20, 20 );
                        var m = new THREE.MeshLambertMaterial( { color: 0xFF91FF
    } );
                        player_meshes[json.data.name] = new THREE.Mesh( g, m );
                        scene.add( player_meshes[json.data.name] );
                }
                all_players[json.data.name] = json.data.coords;
        }
        ⋮
};
```

Listing A.16: This code determines whether the player whose information is being received has been seen before and creates a mesh accordingly. It then places the player mesh at the received coordinates.

```
connection.onmessage = function (message) {
        ⋮
        if (json.type === 'player_kd') {
                if (json.data.name != "false") {
                        all_players_kd[json.data.name] = json.data.kd;
                        display_player_stats();
                }
        }
        ⋮
};
```

Listing A.17: On receipt of a *player_kd* message, the information is stored in an associative array with the player name as its index and the scoreboard is updated. This snippet assumes an *Array()* with the name *all_players_kd* has been defined.

```
function init_player_stats () {
        ⋮
        var score = document.createElement('div');
        score.id = 'player_score';
        score.style.position = 'absolute';
        score.innerHTML = "<center>SCORE</center>";
        score.style.top = 100 + 'px';
        score.style.left = '50%';
        score.style.bottom = '300px';
        score.style.width = '50%';
        score.style.height = '50%';
       score.style.transform = 'translate(-50%, 0)';
        score.style.display = 'none';
        score.style.fontFamily = 'helvetica';
        score.style.fontSize = 32 + 'px';
        score.style.textShadow = "\
                -1px -1px 0 #000,\
                1px -1px 0 #000,\
                -1px 1px 0 #000,\
                1px 1px 0 #000";
        score.style.color = "#AAAAAA";
        score.style.background = "rgba(100,100,100,0.5)";
        var player_kd = document.createElement('table');
        player_kd.id = 'player_kd';
        player_kd.width = '100%';
        player_kd.innerHTML = "<tr width=\"100%\"><th
    width=\"50%\">Player</th><th width=\"25%\">Kills</th><th
    width=\"25%\">Deaths</th></tr>";
        player_kd.style.fontFamily = 'helvetica';
        player_kd.style.fontSize = 16 + 'px';
        player_kd.style.textShadow = "\
                -1px -1px 0 #000,\
                1px -1px 0 #000,\
                -1px 1px 0 #000,\
                1px 1px 0 #000";
        player_kd.style.color = "#DDDDDD";
        score.appendChild(player_kd);
        document.body.appendChild(score);
}

function display_player_stats () {
        ⋮
                stats = document.getElementById('player_kd');
                stats.innerHTML = "<tr width=\"100%\"><th
    width=\"50%\">Player</th><th width=\"25%\">Kills</th><th
    width=\"25%\">Deaths</th></tr>";
                for (var x in all_players_kd){
                        if (all_players_kd.hasOwnProperty(x) && x != "false"){
                                stats.innerHTML += "<tr width=\"100%\"><td
    width=\"50%\">" + x + "</td><td width=\"25%\">" + all_players_kd[x][0] +
    "</td><td width=\"25%\">" + all_players_kd[x][1] + "</td></tr>";
                        }
                }
        ⋮
}
```

Listing A.18: The Scoreboard is dynamically initialised and updated alongside the player information that is displayed in the top-left of the screen.

```
var tabDown = false;
$(document).keydown(function (event) {
        .
        .
        .
        if (event.which == 9) {
                if(!tabDown) {
                        tabDown = true;
                        $(document.getElementById("player_score")).toggle();
                }
                event.stopImmediatePropagation();
                event.stopPropagation();
                event.preventDefault();
        }
});

$(document).keyup(function (event) {
        if (event.which == 9) {
                if(tabDown) {
                        tabDown = false;
                        $(document.getElementById("player_score")).toggle();
                }
                event.stopImmediatePropagation();
                event.stopPropagation();
                event.preventDefault();
        }
        .
        .
        .
});
```

Listing A.19: To display the Scoreboard, the player must hold down the *Tab* key. This is in line with standard first-person shooter gameplay.

```
<body>
        <canvas id="text" style="display: none;"></canvas>
        ⋮
</body>
<!-- Player Name as Texture Init -->
<script>
        var text_canvas = document.getElementById("text");
        var ctx = text_canvas.getContext("2d");
        var textSize = 12;
        ctx.font = textSize+"px monospace";     // This determines the size of
    the text and the font family used

        function getPowerOfTwo(value, pow) {
                var pow = pow || 1;
                while(pow<value) pow *= 2;
                return pow;
        }

        function get_player_name_texture (name) {
                text_canvas.width = getPowerOfTwo(ctx.measureText(name).width);
                text_canvas.height = getPowerOfTwo(textSize*2);
                ctx.fillStyle = "#FFFFFF";      // This determines the text
    colour, it can take a hex value or rgba value (e.g. rgba(255,0,0,0.5))
                ctx.textAlign = "center";       // This determines the alignment
    of text, e.g. left, center, right
                ctx.textBaseline = "middle";    // This determines the baseline
    of the text, e.g. top, middle, bottom
                ctx.fillText(name, text_canvas.width/2, text_canvas.height/2);
                var canvasTexture = new THREE.Texture(text_canvas);
                canvasTexture.needsUpdate = true;
                return canvasTexture;
        }
</script>
```

Listing A.20: This snippet initialised the canvas and defines the function that will be called every time the name on the canvas needs to be changed.

```
if (json.data.name != "false" && json.data.name != myName && new_player) {
        var tex = get_player_name_texture(json.data.name);
        g = new THREE.PlaneGeometry( text_canvas.width, text_canvas.height);
        m = new THREE.MeshBasicMaterial( { map: tex, transparent: true } );
        m.side = THREE.DoubleSide;
        player_name_meshes[json.data.name] = new THREE.Mesh( g, m );
        scene.add( player_name_meshes[json.data.name] );
}
```

Listing A.21: Player nametags are initialised at the same time player models are initialised. This assumes an *Array()* with the name *player_name_meshes* has been defined.

```
function draw_players (ps) {
        for (var x in ps){
                if (ps.hasOwnProperty(x) && x != myName && x != "false"){
                        player_meshes[x].position.set(ps[x][0], ps[x][1],
    ps[x][2]);
                        player_name_meshes[x].position.set(ps[x][0],
    ps[x][1]+20, ps[x][2]);
                        player_name_meshes[x].rotation.x =
    controls.getObject().rotation.x;
                        player_name_meshes[x].rotation.y =
    controls.getObject().rotation.y;
                        player_name_meshes[x].rotation.z =
    controls.getObject().rotation.z;
                }
        }
}
```

Listing A.22: Player nametags are drawn at the same time player models are drawn. The nametags are drawn to rotate alongside the player camera rotation so that the name will always be directly facing the player.

```
var timeout;
var onMouseDown = function ( event ) {
        timeout = setInterval(function(){
                var vector = new THREE.Vector3( 0 , 0 , -1 );
                vector.unproject(camera);
                var rayCam = new THREE.Ray(controls.getObject().position,
    vector.sub(controls.getObject().position).normalize() );
                var rayCaster = new THREE.Raycaster(rayCam.origin,
    rayCam.direction);

                var player_meshes_array = [];
                var player_names_array = new Array();
                for (var x in player_meshes){
                        if (player_meshes.hasOwnProperty(x) && x != "false"){

    player_meshes_array[player_meshes_array.length++] = player_meshes[x];
                                player_names_array[player_meshes[x].id] = x;
                        }
                }

                var intersects = rayCaster.intersectObjects(player_meshes_array);

                if (intersects.length) {
                        for (var x in player_names_array){
                                if (player_names_array.hasOwnProperty(x) && x !=
    "false" && parseInt(x) === intersects[0].object.id){
                                        player_shots.push(player_names_array[x]);
                                }
                        }
                }
        }, 50);
};
```

LISTING A.23: In order to find out the name of the player shot, an array of player names is created with the mesh id as the index. This mesh id is then used to access the player name when pushing the shot to the shot queue. This assumes the existence of a *player_shots Array()*

```
connection.onmessage = function (message) {
        ⋮
        if (json.type === 'player_shot') {
                console.log(json.data.shooter + " shot " + json.data.shootee);
                if (json.data.shootee === myName) {
                        myHealth--;
                        if (!myHealth) {
                                console.log("You were killed by " +
    json.data.shooter);
                                player_killed_by = json.data.shooter;
                        }
                        display_player_stats();
                }
        } else if (json.type === 'player_kill') {
                console.log(json.data.killer + " killed " + json.data.killee);
                if (json.data.killer === myName) {
                        myKills++;
                        display_player_stats();
        }
}

setInterval(function() {
        ⋮
        while (player_shots.length) connection.send(JSON.stringify({type:
    'player_shot', shot: player_shots.shift()}));
        if (player_killed_by != false) {
                connection.send(JSON.stringify({type: 'player_death', killer:
    player_killed_by}));
                player_killed_by = false;
        }
}, 10);
```

Listing A.24: The messages sent and received by the client involved in the act of shooting, killing or dying.

Bibliography

[1] Khronos Group. OpenGL ES 2.0 for the Web, 2015. URL https://www.khronos.org/webgl/.

[2] Kaazing Corporation Peter Lubbers & Frank Greco. HTML5 Web Sockets: A Quantum Leap in Scalability for the Web, 2013. URL https://www.websocket.org/quantum.html.

[3] Google. Chrome Experiments - WebGL Experiments, 2015. URL https://www.chromeexperiments.com/webgl.

[4] David Li. Volumentric Particle Flow, 2015. URL http://david.li/flow/.

[5] Awwwards. The awards for design, creativity and innovation on the Internet - WebGL Category, 2015. URL http://www.awwwards.com/websites/webgl/.

[6] Reebok. Gray Matters: An Active Body Is An Active Mind, 2014. URL http://fitness.reebok.com/international/be-more-human/#/page/gray-matters.

[7] Little Workshop. BrowserQuest, 2012. URL http://browserquest.mozilla.org/.

[8] Paul Rouget. BrowserQuest – a massively multiplayer HTML5 (WebSocket + Canvas) game experiment, 2012. URL https://hacks.mozilla.org/2012/03/browserquest/.

[9] The Ironbane Team. Ironbane Alpha, 2015. URL http://play.ironbane.com/.

[10] "Nikke". 0.4 released!, 2015. URL http://ironbane.com/topic/240/0-4-released.

[11] Diegon Cantor & Brandon Jones. *WebGL Beginner's Guide*. PACKT, Birmingham, UK, 2012.

[12] Toni Parisi. *WebGL Up and Running*. O'Reilly, Sebastopol, CA, 2012.

[13] Ricardo Cabello. three.js - JavaScript 3D library, 2015. URL http://threejs.org/.

[14] Emil Persson. Humus - Textures, 2014. URL http://www.humus.name/index.php?page=Textures&ID=77.

[15] N. Greene. Environment mapping and other applications of world projections. *Computer Graphics and Applications, IEEE*, 6(11):21–29, November 1986. URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4056759.

[16] "kripken". ammo.js/examples/webgl_demo/, 2014. URL https://github.com/kripken/ammo.js/tree/master/examples/webgl_demo.

[17] Martin Sikora. Node.js & WebSocket - Simple chat tutorial, 2011. URL http://ahoj.io/nodejs-and-websocket-simple-chat-tutorial.

[18] Soledad Penadés. Object picking, 2014. URL http://soledadpenades.com/articles/three-js-tutorials/object-picking/.

[19] Joe Hewitson. Three.js and Babylon.js: a Comparison of WebGL Frameworks, 2013. URL http://www.sitepoint.com/three-js-babylon-js-comparison-webgl-frameworks/.

[20] Babylon.js team. WEBGL. simple. powerful., 2013. URL http://www.babylonjs.com/.

[21] Mozilla. Unity 5 Ships and Brings One Click WebGL Export to Legions of Game Developers, 2015. URL https://blog.mozilla.org/blog/2015/03/03/unity-5-ships-and-brings-one-click-webgl-export-to-legions-of-game-developers/.

[22] Florian Boesch. Why you should use WebGL, 2013. URL http://codeflow.org/entries/2013/feb/02/why-you-should-use-webgl/.

[23] Khronos Group. WebGL 2 Specification, 2015. URL https://www.khronos.org/registry/webgl/specs/latest/2.0/.

[24] "mbest". An Early Look at WebGL 2, 2015. URL https://blog.mozilla.org/futurereleases/2015/03/03/an-early-look-at-webgl-2/.